

Move to the Cloud

Lab version: 1.2.0

Last updated: 2/5/2014

# Overview

Modern line-of-business applications have much to gain from Windows Azure. In addition to the increased scalability and reliability the cloud has to offer, cloud-based services are also easier to access from outside the corporate firewall. This new accessibility opens a whole class of opportunities for developers to enhance their functionality and extend their user experiences to new devices and tablets.

In our scenario, we have a variety of resources and services that exist behind a firewall. Now that we want to extend their reach and functionality, the time has come to move the experience into the cloud. The easiest way for us to migrate the project to the cloud is piece-by-piece. We’ll start off with our functional system, and then move just the database. We’ll be able to confirm success by wiring the existing on-premises WCF service to the new in-cloud database, and everything else should continue to work as expected. After that, we’ll move the service itself out to the cloud. By changing the WCF service URL configured in the WPF application, it should also immediately work. To secure the WCF service, we’ll extend the reach of our on-premises Active Directory out to the cloud using Windows Azure Active Directory, which will enable us to require users of the WCF service to be authenticated first. Finally, we’ll extend the cloud-based WCF service by integrating it with an on-premises resource using Service Bus Relay, a mechanism that allows us to securely reach inside the firewall.

## Important Note

Many of the screenshots in this lab use configuration fields from Windows Azure that are globally unique. As a result, your settings will likely be different. Please take care to use the fields for your configuration. For example, the domains **“expenseswcf.azurewebsites.net”** and **“expenses.onmicrosoft.com”** are used in the screenshots, but you should use the domain names you generate during your session. Other places where screenshots will vary include the SQL Server name and administrator name, Windows Azure publish settings, and various GUIDs. Effort has been made to call these out in the steps, but please double-check copies & pastes.

## Objectives

In this hands-on lab, you will learn how to:

* Migrate an on-premises database to SQL Azure
* Migrate an on-premises WCF service to Windows Azure Web Sites
* Secure the WCF service using Azure Active Directory
* Integrate the WCF service with an on-premises service using Service Bus Relay

## Prerequisites

The following is required to complete this hands-on lab:

* Microsoft Visual Studio 2013
* The latest Windows Azure SDK for .NET
* A Windows Azure subscription

## Setup

This lab picks up where module 2 left off.

## Exercises

This hands-on lab includes the following exercises:

1. Deploying to SQL Azure
2. Deploying to Windows Azure Web Sites
3. Securing a WCF service using Windows Azure Active Directory
4. Integrating a cloud service with on-premises resources using Service Bus Relay

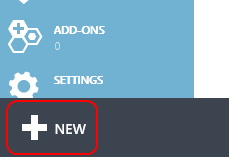
# Exercise 1: Deploying to SQL Azure

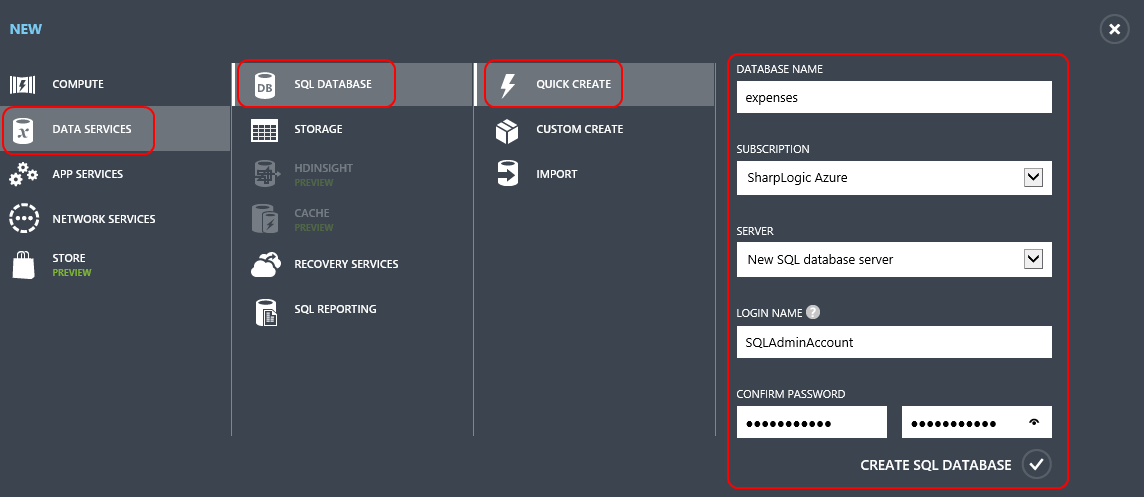
In this exercise, we’ll go through the process of setting up a SQL Azure database server. Then we’ll deploy our existing database, including its data, out to that server.

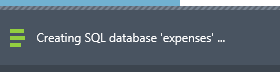
## Task 1: Creating a SQL Azure database

In this task, we’ll create a SQL Azure database.

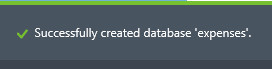
1. Log into your Azure administrative account at <https://manage.windowsazure.com>.
2. In the bottom right corner, click the **New** button.



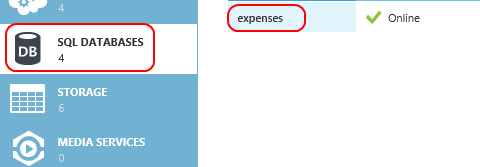
1. Click **Data Services | SQL Database | Quick Create** and fill out the form to create your database. If you already have an existing SQL server configured in Azure you’ll see the option to reuse it. For our exercise, please select the option to create a new one.
2. It will take a few moments for your database to get created.



1. Once the empty database has been created, you’ll see a message like this one.



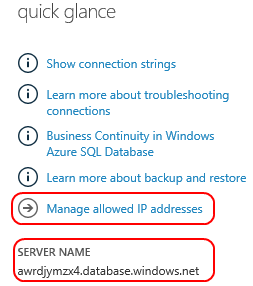
1. Click the **SQL Databases** tab along the left edge of the page and click on the newly created **expenses** database. Note that you’ll need to click on the name specifically to navigate to its page.



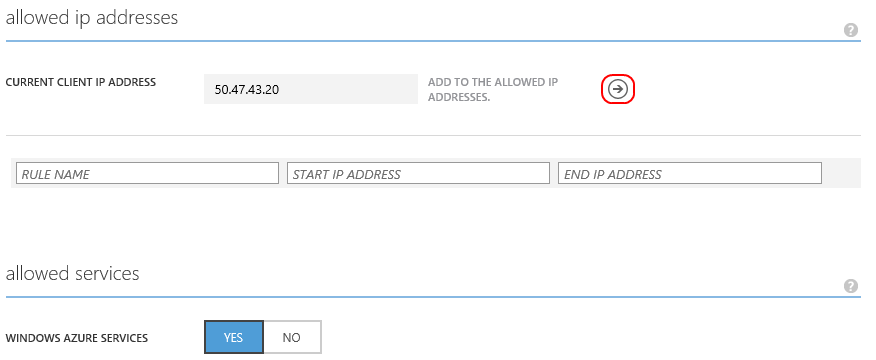
1. Click the **Dashboard** link to see more options for the database.



1. In the **Quick Glance** section along the right edge you can get some useful information about your server. Take note of the **Server Name**, which we’ll need later on for deploying the database. By default, only IP addresses inside the Azure datacenter have access to the database server. Next, click the **Manage allowed IP addresses** link to allow your current connection to access the database as well.



1. If the public IP you connect from is a static IP address, you can add it by clicking the **Add to the allowed IP addresses** button. If you’re accessing via network with a range of public IPs, you can add a rule for that range. Note that you also have the option to disallow access to your server from services running inside Azure, which may be useful if all access comes from external sources, only. In our case, we will be ultimately connecting to this database via services running in Azure, so we’ll leave the option enabled.



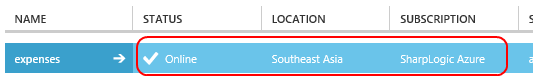
1. After adding any necessary rules, click the **Save** button at the bottom of the page to commit them.



1. Click the **Databases** link to view the databases on this server.



1. Click the row of the **expenses** database, but not on the name itself. We want to select the row, not navigate to the database’s record.



1. Click the **Delete** button at the bottom of the page to delete this database.

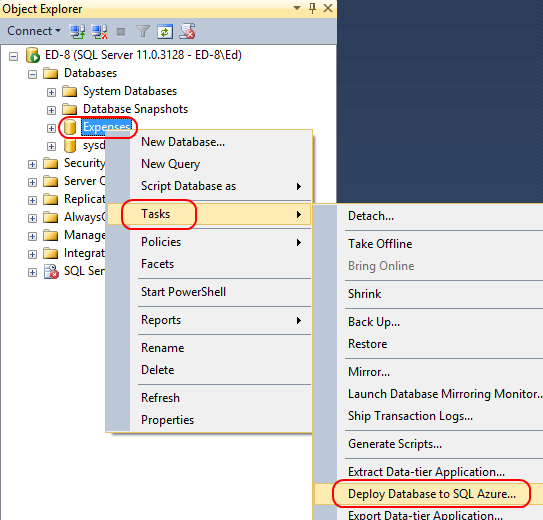


1. The management tool will ask you to confirm that you want to delete the database. Accept the confirmation. However, it will then ask you if you would like to delete the server because it no longer has any databases. Deny this request since we’ll be deploying a database in the next step.

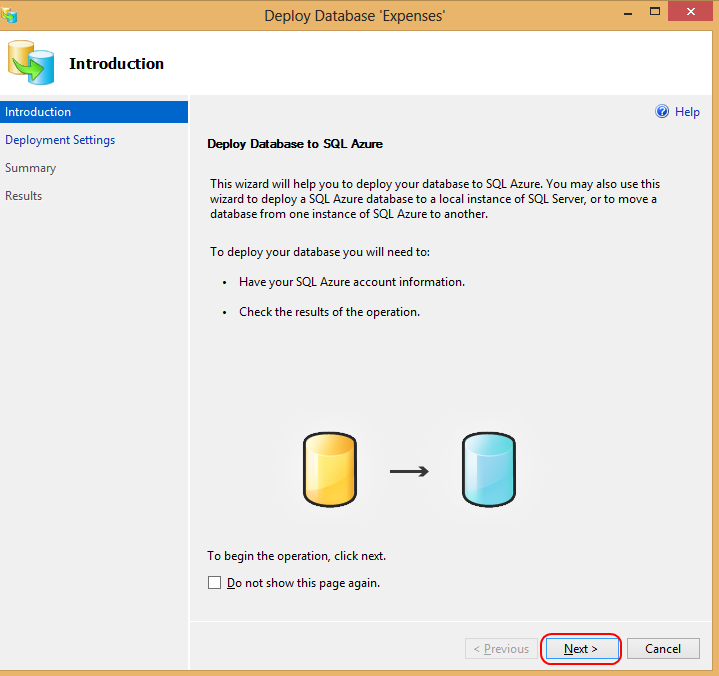
## Task 2: Deploy your existing database to SQL Azure

In this task, we’ll deploy your existing database to SQL Azure.

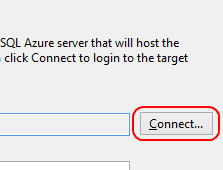
1. Open **SQL Server Management Studio**.
2. Connect to the server containing your local **Expenses** database.
3. In the **Object Explorer**, right-click the **Expenses** database and select **Tasks | Deploy Database to SQL Azure…**.



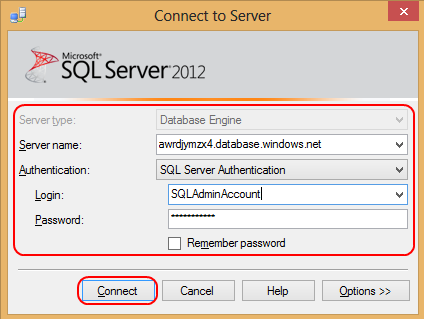
1. If presented with the **Introduction** step of the wizard, click **Next** to continue.



1. On the **Deployment Settings** page, click **Connect** to connect to the target server.

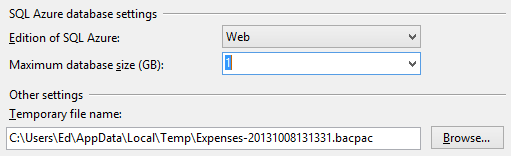


1. Fill out the connection details for **your SQL Azure server** using the settings configured during creation earlier and click **Connect.**



Note that there are some options available for configuration. In the **SQL azure database settings**, we can decide if we want to use the **Web** or **Business** editions of SQL Azure. At this time, the only difference between the two is the maximum database sizes they allow. Web is limited to up to 5GB, whereas Business is intended for databases up to 150GB. For the purposes of our lab, we can use the smallest option available.

In the **Other settings**, you can decide where the temporary bacpac file is stored. If you have a very large database with limited space on your default drive, this gives you the option to select a different path.



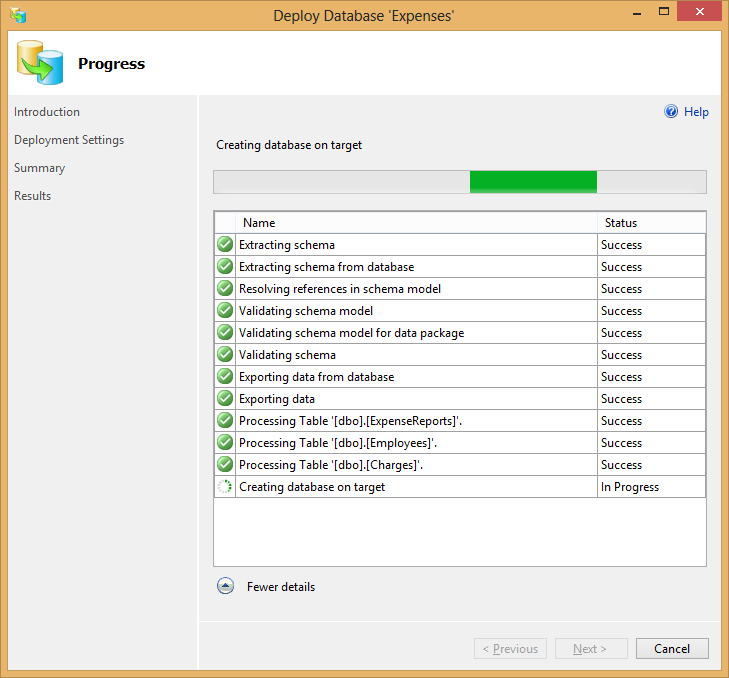
1. Once the connection succeeds, click **Next** to continue.



1. Review the **Summary** page. Click **Finish** to begin the deployment.



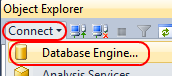
1. It’ll take a minute or two for the database to fully deploy. During the operation you can see the steps it takes.



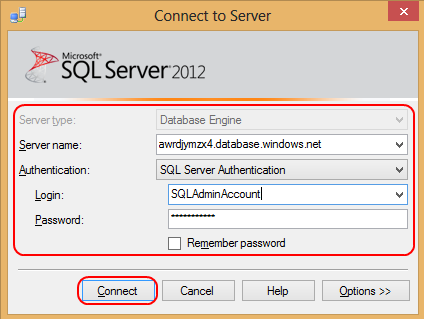
1. Click **Close** when complete.



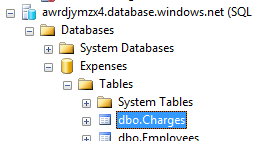
1. In the **Object Explorer**, click **Connect | Database Engine…**.



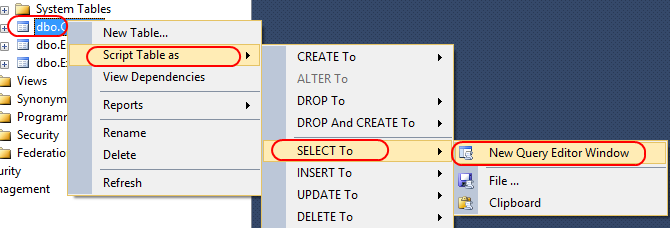
1. Connect to **your SQL Azure server**.



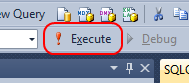
1. Expand **[your server] | Databases | Expenses | Tables**.



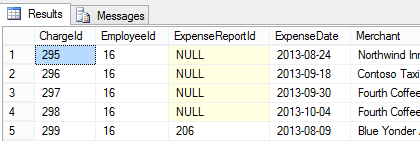
1. Right-click the **Charges** table. Note that the options when working with SQL Azure are somewhat limited versus those available for a local SQL instance. SQL Azure still supports almost everything local SQL does, except that some of the features are not exposed via the tool and require scripting. For example, there is no option to edit records inline. Select **Script Table as | SELECT To | New Query Editor Window**.



1. Click **Execute** to run the query.



1. We can now see that all of our data is available in our SQL Azure database.



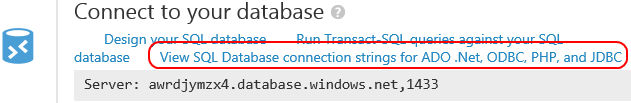
## Task 3: Updating the Expenses WCF service to use the new database

In this task, we’ll update our WCF service to use the newly created SQL Azure database.

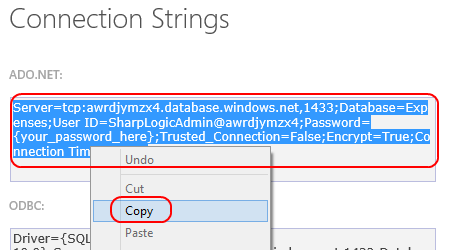
1. Log into your Azure administrative account at <https://manage.windowsazure.com> if it’s not already open.
2. Navigate to the newly deployed database by clicking the **SQL Databases** tab, followed by the **Expenses** database link.



1. In the **Connect to your database** section, click the **View SQL Database connection strings for ADO.NET, ODBC, PHP, and JDBC** link.



1. In the **Connection Strings** dialog, highlight and copy the connection string from the **ADO.NET** box. Press **Esc** to close the dialog when done.



1. Open the **Web.config** file at **\Expenses WCF Service\Expenses.WcfService\Web.config**. You can use any text editor, whether it’s Visual Studio 2013 or Notepad.
2. In the **ConnectionStrings** section, change **Expenses.WcfService.ServiceCore.Properties.Settings.ExpensesConnectionString** to use the new connection string for your SQL Azure database. Be sure to put your password in where specified.
3. Save **Web.config**.
4. Run the **Expenses** client application to confirm that it works as expected, but is now using data from SQL Azure.

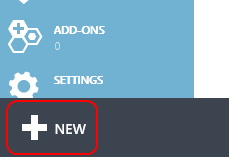
# Exercise 2: Deploying to Windows Azure Web Sites

In this exercise, we’ll go through the process of setting up a Windows Azure Web Site. Then we’ll deploy our existing WCF service out to it.

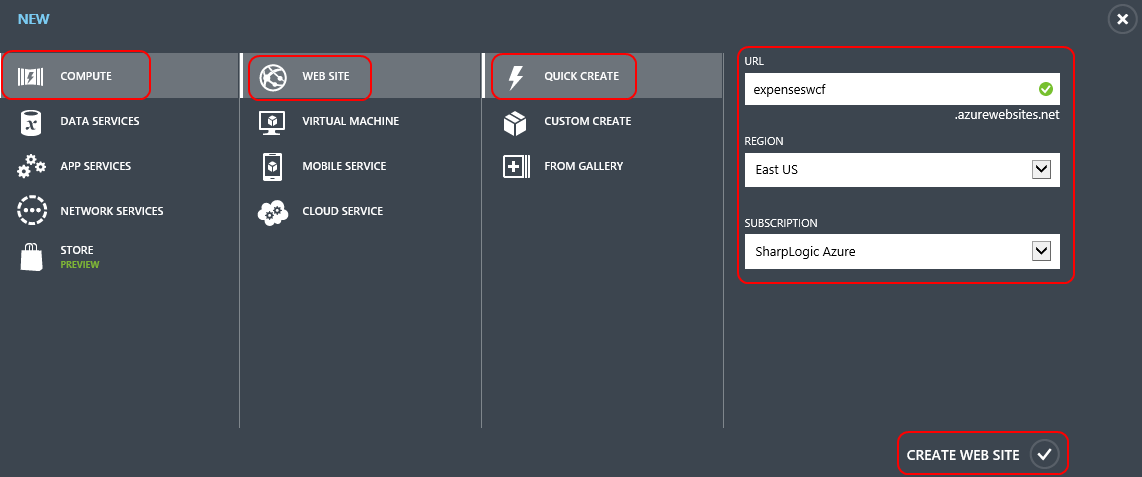
## Task 1: Creating Windows Azure Web Site

In this task, we’ll create a Windows Azure Web Site.

1. Log into your Azure administrative account at <https://manage.windowsazure.com> if not already open.
2. Click the **New** button in the bottom left corner.



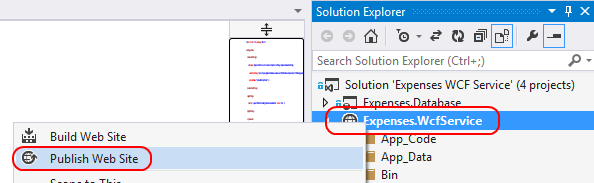
1. Select **Compute | Web Site | Quick Create** and fill out the settings. You’ll need to come up with a unique name for the **URL**. Your eventual domain name will be **[your URL name].azurewebsites.net**. Click **Create Web Site** to create the Web site.



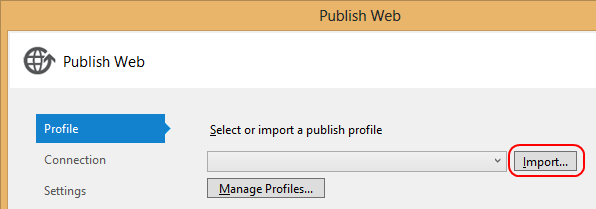
## Task 2: Deploy the Expenses WCF service to our new Windows Azure Web Site

In this task, we’ll update our WCF service to use the newly created SQL Azure database.

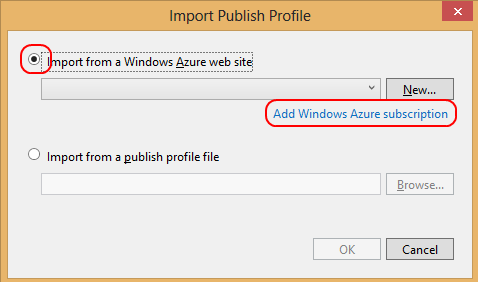
1. After the Web site has been created, open **\Expenses WCF Service\Expenses WCF Service.sln** in Visual Studio 2013.
2. Right-click **Expenses.WcfService** in **Solution Explorer** and select **Publish Web Site**.



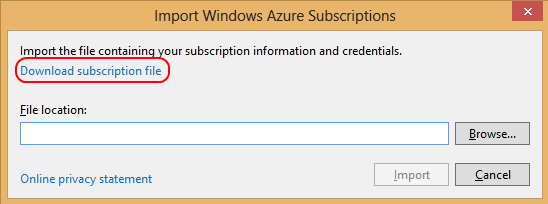
1. In the **Publish Web** dialog, click **Import…**. Once we have this profile set up, we can use it in the future and jump right to the deployment step. For this lab, we’ll go through the whole process of downloading and importing the publish profile for our Windows Azure subscription.



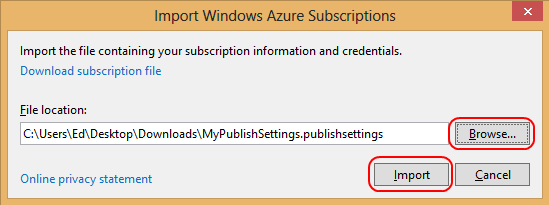
1. In the **Import Publish Profile** dialog, select the **Import from a Windows Azure web site** option and click the **Add Windows Azure subscription** link.



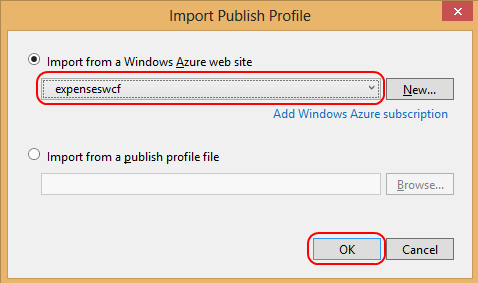
1. In the **Import Windows Azure Subscriptions** dialog, click the **Download subscription file** link. This will open a browser window to the Azure site. Save the file being automatically downloaded to disk somewhere convenient, such as the desktop.



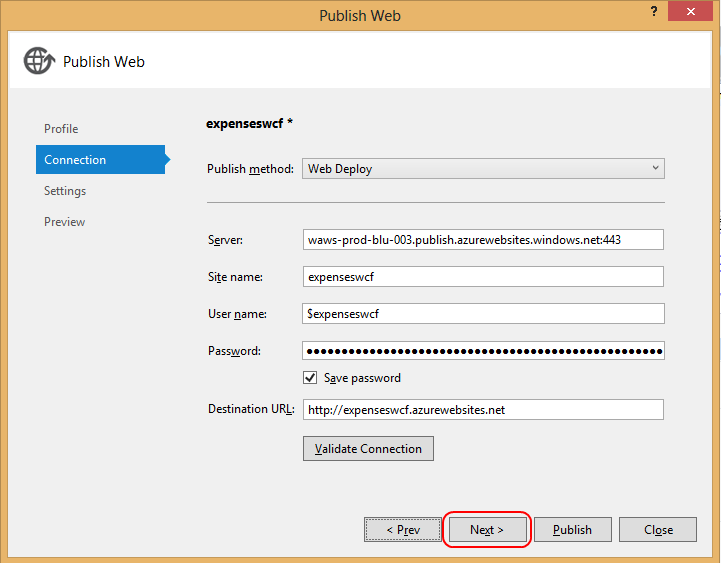
1. Click **Browse…** and select the file you just downloaded. Click **Import** to complete the import process.



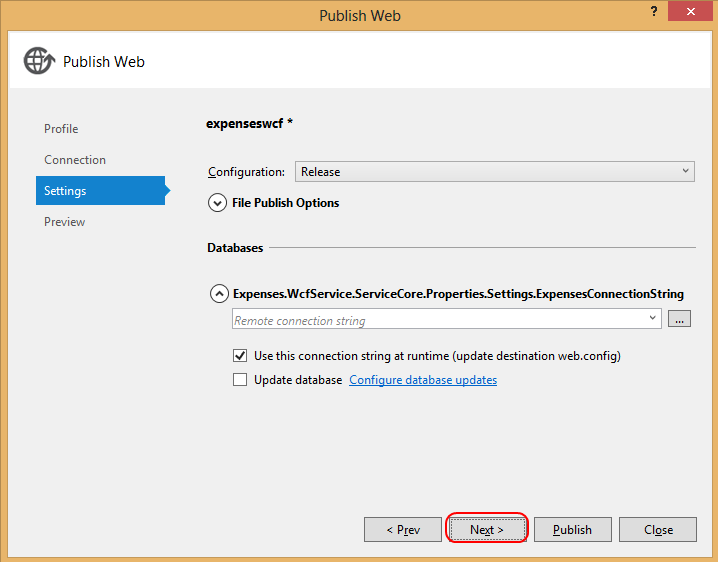
1. In the **Import Publish Profile** dialog, select the site you created earlier (that was imported in the previous step). Click **OK**.

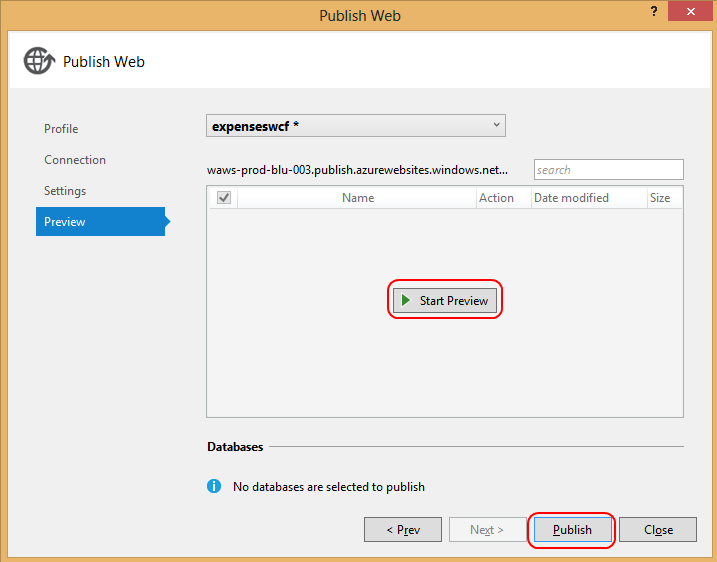


1. In the **Connection** step of the **Publish Web** wizard, click **Next** to continue.



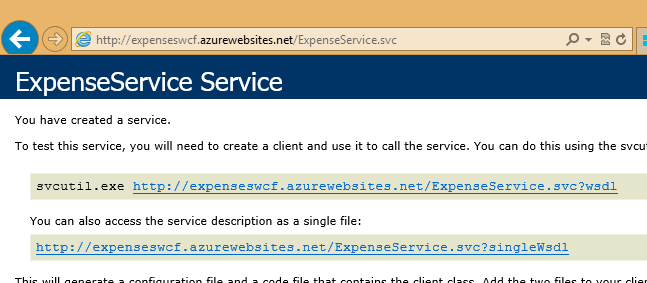
1. In the **Settings** step of the **Publish Web** wizard, click **Next** to continue.



1. In the **Preview** step of the **Publish Web** wizard, click **Start Preview** to see what files have changed and need to be deployed. Since this is our first deployment, it will be all of them. Note that it will only show a few because only the built files that are necessary to run the service are published. Click **Publish** to publish.
2. Once the deployment is complete, a browser window will open to the domain root. In our case, we don’t have a default page, so one like this will be shown in its place.



1. To confirm our service is available, add “ExpenseService.wcf” to the end of the URL and press **Enter**. You should see the service default page.



## Task 3: Updating the Expenses WPF client to use the new service

In this task, we’ll update our WPF client to use the new cloud-hosted WCF service.

1. Open **\Expenses\Expenses.sln** in Visual Studio 2013. Use a new instance of Visual Studio so that both the service and client solutions are open.
2. From the **Expenses.Wpf** project, open **App.config**.
3. In the **configuration/appSettings** section, change the **expenseServiceUrl** to match the domain of your new hosted service. It should look like ***http://[site\_name\_from\_earlier].azurewebsites.net/ExpenseService.svc***. It may be easiest to copy the URL used in the previous task to confirm the service deployment.

Note that for any app already deployed across our enterprise, we could just update the config files they use locally to reach this new service endpoint.

1. Press **F5** to launch the application. It should connect to the cloud-hosted service and work as expected.

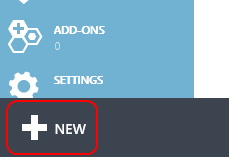
# Exercise 3: Securing a WCF service using Windows Azure Active Directory

In this exercise, we’ll secure our WCF service using Windows Azure Active Directory.

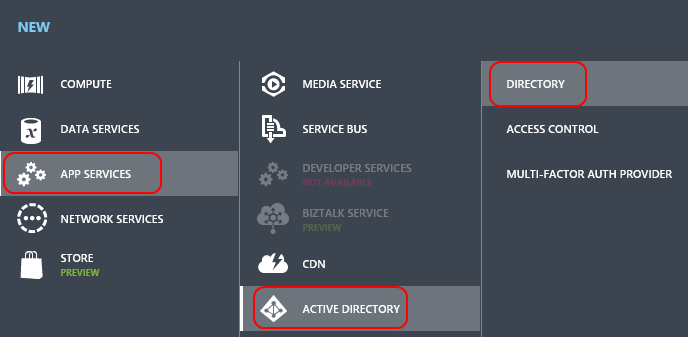
## Task 1: Creating a Windows Azure Active Directory tenant and add a user

In this task, we’ll create a Windows Azure Active Directory tenant. We’ll also add a user, which is specific to this lab. In a real environment, we would set up Active Directory synchronization so that our Azure AD instance relied on the users pushed from the on-premises instance.

1. Log into your Azure administrative account at <https://manage.windowsazure.com> if not already open.
2. In the bottom right corner, click the **New** button.



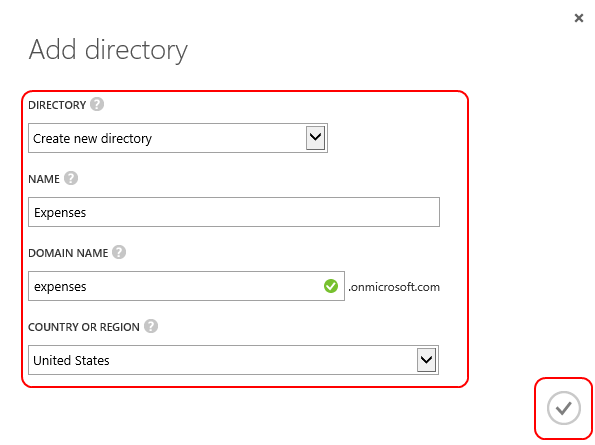
1. Select **App Services | Active Directory | Directory**.



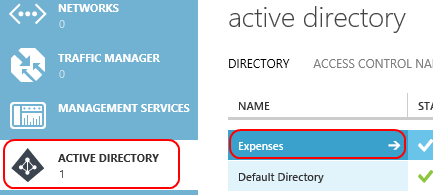
1. Click **Custom Create** to open the **Add directory** dialog.



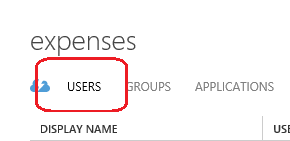
1. In the **Add directory** dialog, select the option to create a new directory and fill out the form. Note that you’ll need to provide a unique name for the **Domain Name** field. Click the **Create** check button when complete. Note that the domain name you select here will be used later when securing the WCF application. In the example screenshot below, the value is **expenses.onmicrosoft.com**.



1. Click the **Active Directory** tab to view the directories associated with your account. Click the newly created directory to view it.



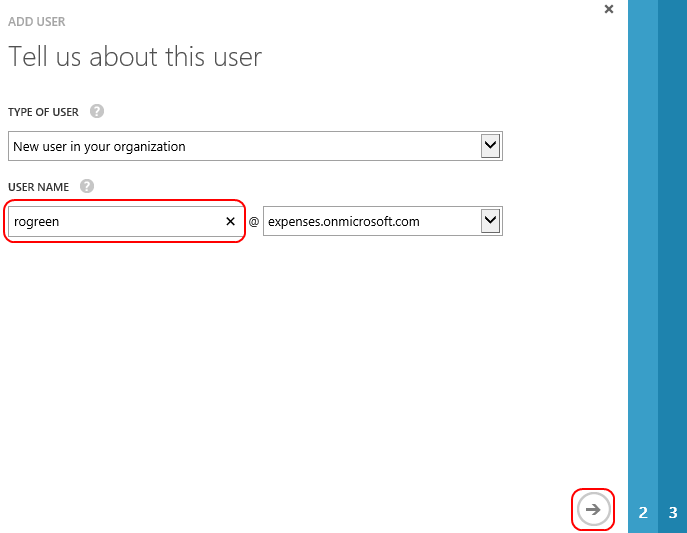
1. Click **Users** at the top of the screen.



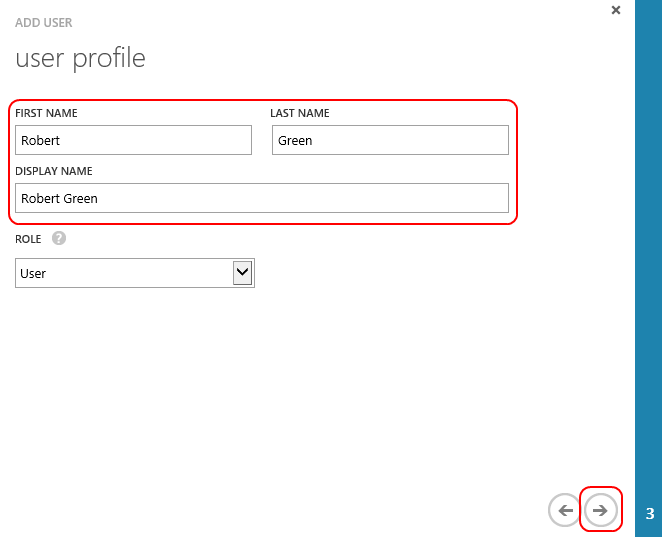
1. In the bottom menu, click **Add user**.



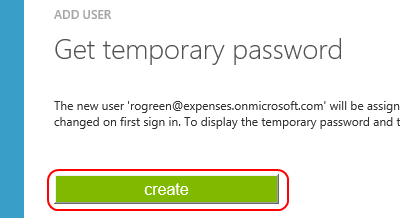
1. Enter “rogreen” as the **User Name** and click the **Next** arrow.



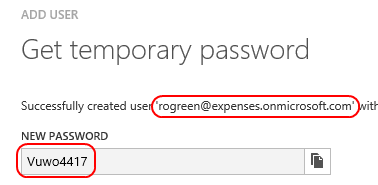
1. Fill out the form with Robert Green’s information and click the **Next** arrow.



1. Click **create** to create the account and view the temporary password.



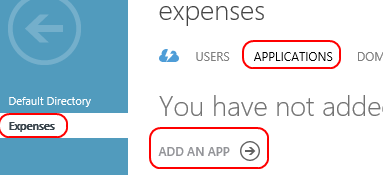
1. Make note of the email address and password created for this user. You may want to copy them into notepad for temporary storage. Note that your email address and password will very likely be different from the ones shown in the screenshot below. Press **Esc** to close the dialog when complete.



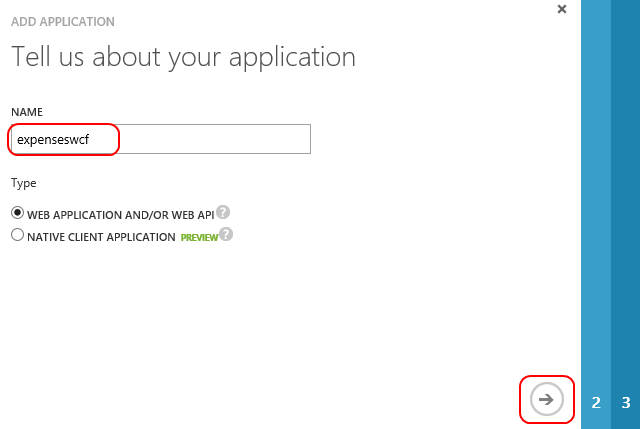
## Task 2: Creating a Windows Azure Active Directory service application

In this task, we’ll create a Windows Azure Active Directory service application that will allow our WCF service to authenticate requests. Then we’ll update our service so that it requires authorization before it will process service requests.

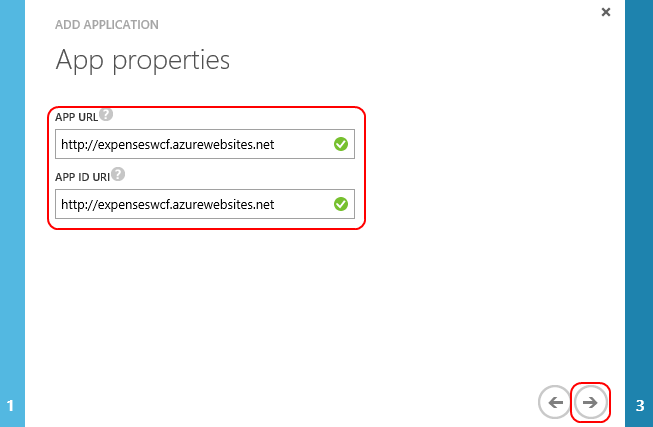
1. Click the **Expenses** link on the left side of the main view. Then click **Applications** to view the applications associates with this tenant. We don’t have any yet, so click **Add an app** to create our first app.



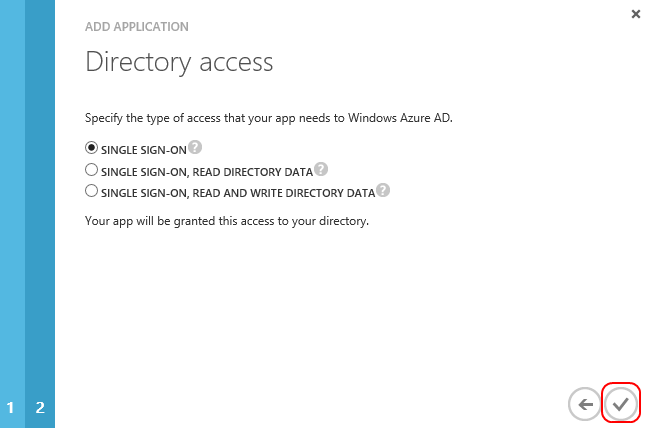
1. Enter “Expenses WCF” as the **Name** of your application and click the **Next** arrow.



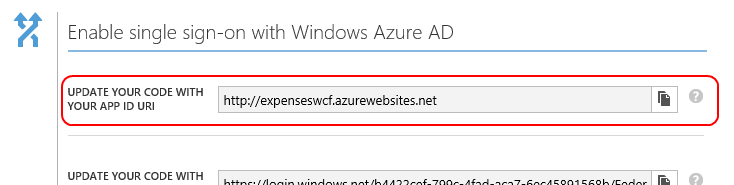
1. On the **App properties** page, use the URL of the root of your Azure Web Site as the **App URL** and **App ID URI**. Click the **Next** arrow to continue.



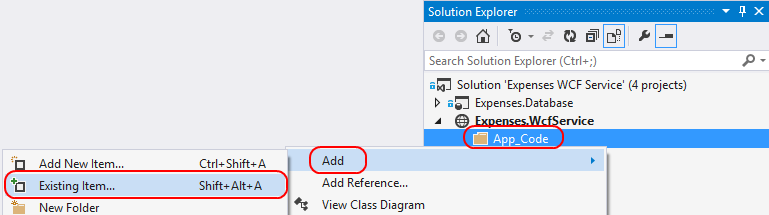
1. On the **Directory access** page, use the default “Single sign-on” option and click the **Finish** check.



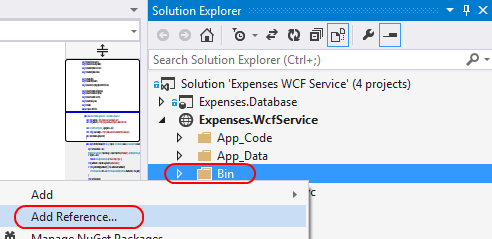
1. In the middle of the **Quick Start** page you can see some useful details about your tenant’s application. The main item of interest for us is the **App ID URI**, which we set earlier as the Azure Web Site root. We’ll need to use this to configure our WCF service.



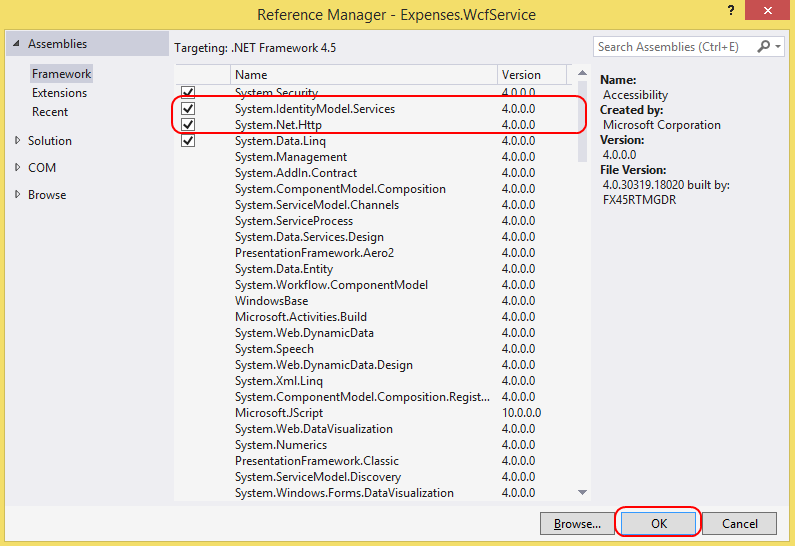
1. Bring up the Visual Studio 2013 instance with the WCF service open. We’re going to need to update our service to look for an authorization token in future WCF requests. If that token doesn’t exist, we’ll need to return an Unauthorized response with details on where the token can be acquired. Fortunately, this functionality is already built out and we can simply add it into our project. In the **Solution Explorer** under the **Expenses.WcfService** project, right-click the **App\_Code** directory and select **Add | Existing Item…**. Navigate to the BearerTokenMessageInspector.cs file provided with this lab and add it.



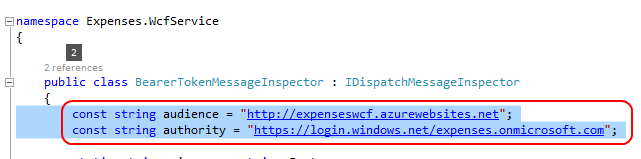
1. We’ll need to add some references to support out new class. Right-click the **Bin** directory and select **Add Reference…**.



1. Navigate through the list to add the 4.0 versions of **System.IdentityModel.Services** and **System.Net.Http**. Click **OK** when done.



1. Select **Tools | Library Package Manager | Package Manager Console**.
2. In the **Package Manager Console** dialog, type “Install-Package System.IdentityModel.Tokens.Jwt –version 1.0.0” and press **Enter**. This will install the JSON Web Token Handler For the Microsoft .NET Framework 4.5 NuGet package. This package provides an assembly containing classes that extend the .NET Framework 4.5 with the ability to process the JSON Web Token (JWT) format.
3. In the **App\_Code** directory of our **Expenses.WcfService** project, double-click **BearerTokenMessageInspector.cs** to open it. Change the **audience** member to use the root URL of the Azure Web Site we used to configure most of the fields of the Windows Azure Active Directory tenant applications. In the portal, this is the **App ID URI** field. The **authority** field should be **https://login.windows.net/[your Windows Azure Active Directory tenant domain]**. Note that this is not the Azure Web Sites domain created earlier.



1. Now that our project builds, there is a final step to configure WCF to use this inspector on all requests. We can make these changes in Web.config. First, we’ll need to add our **bearerTokenRequired** node to the active behavior. Please note that it must be the last item in the parent **behavior** node. Second, we’ll need to register our behavior extension with WCF.

<system.serviceModel>

<behaviors>

<serviceBehaviors>

<behavior>

...

**<bearerTokenRequired/>**

</behavior>

</serviceBehaviors>

</behaviors>

**<extensions>**

**<behaviorExtensions>**

**<add name="bearerTokenRequired"**

**type="Expenses.WcfService.BearerTokenExtensionElement, App\_Code"/>**

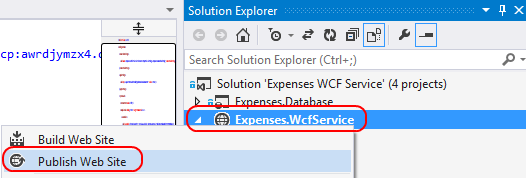
**</behaviorExtensions>**

**</extensions>**

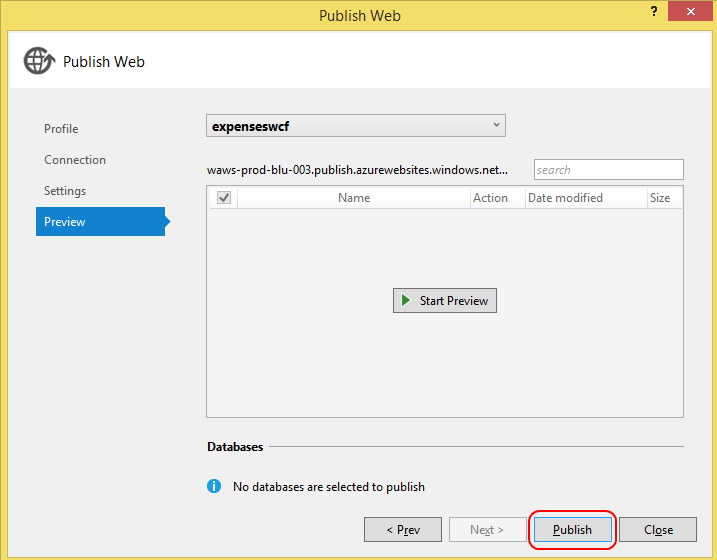
...

</system.serviceModel>

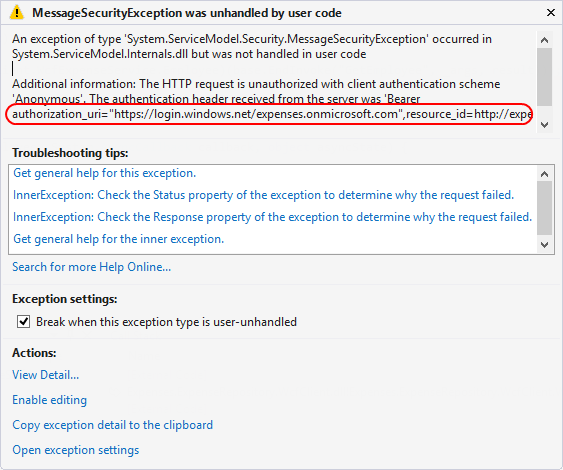
1. Right-click the **Expenses.WcfService** project and select **Publish Web Site**.



1. In the **Publish Web** dialog, click **Publish** to update the public Web site with the latest bits.



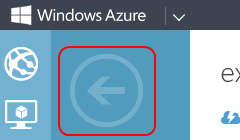
1. In the instance of Visual Studio 2013 that has the **Expenses** WPF project open, press **F5** to run the application. It should throw an exception during initialization that it is not authorized to access the service. If you read the details, you can see that the response from the WCF service is telling our client to authorize at the URL we configured earlier. End the debugging session when ready to continue.



## Task 3: Creating a Windows Azure Active Directory service application

In this task, we’ll create a Windows Azure Active Directory client application that will allow our WPF client to authenticate with Azure AD in order to use our secured WCF service.

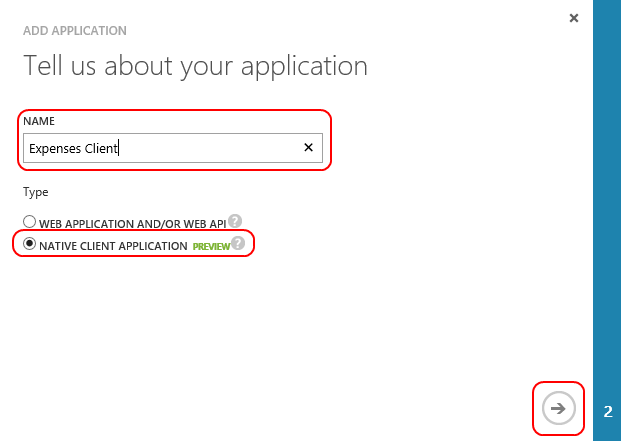
1. Switch back over to the Windows Azure management browser window. Click the big **Back** button in the top left corner of the Window Azure management tool UI. This will bring us up one level to view our Expenses tenant.



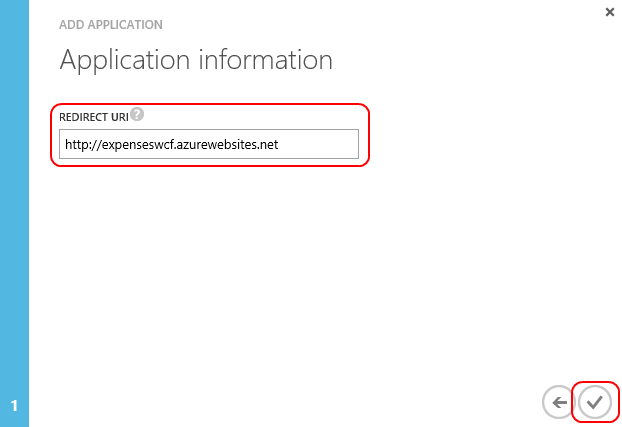
1. In the menu bar at the bottom, click **Add** to create a new application for this tenant.



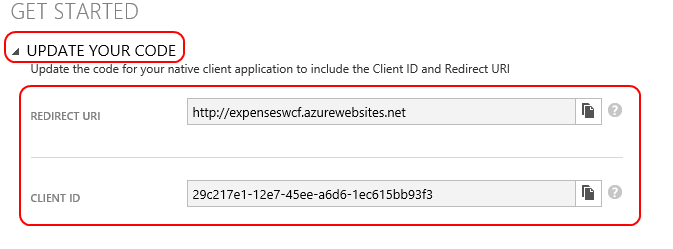
1. Enter “Expenses Client” as the **Name** and select **Native client application** as the **Type**. Click the **Next** arrow button to continue.



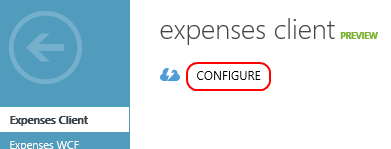
1. On the **Add application** page, use the URL of the root of your Azure Web Site as the **Redirect URI**. Click the **Finish** check when done.



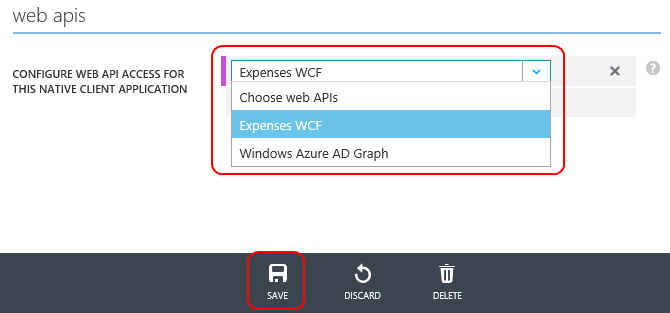
1. Click the **Update your code** expander in the middle of the page. This reveals the two settings we’ll need to configure our client with to connect to the secured service.



1. We’ll also need to configure our service on the portal to grant access to the WCF service. Click **Configure** to view this option.

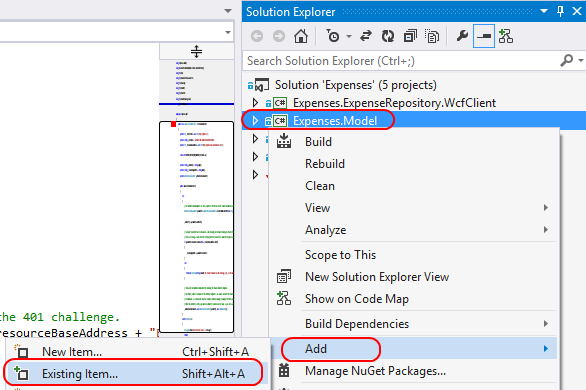


1. In the **Web APIs** section, select the **Expenses WCF** option to allow clients who authenticate with this client application to also be authenticated with that service application. Click **Save** to apply the changes.

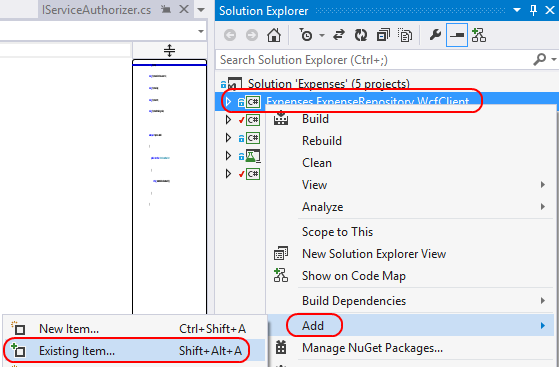


1. Bring up the Visual Studio 2013 instance with the WPF client open. We’re going to need to update the application so that it will acquire a valid token before it makes any WCF service requests. One of the challenges we’re going to face is how we insert our authorization header into all outgoing WCF requests to the Expenses service. Since the repository is factored away into a distinct assembly with no knowledge of our application, we’ll need to perform a little extra work to make it smooth. The first thing we’ll do is to introduce a new interface into our **Model** assembly. This interface will provide a way for the repository to request an authorization header just-in-time during the request pipeline. Later on we’ll add a class that implements this interface at the UI project level.

Right-Click the **Expenses.Model** project and select **Add | Existing Item…**. Navigate to the **IServiceAuthorizer.cs** file provided with this lab and add it to the project.



1. Double-click the **IServiceAuthorizer.cs** file to open it. Note that it only has one method called **GetAuthorizationHeader** that returns a string. Close the file when ready to continue.
2. We’ll also need to update the repository project itself to support this new interface. Right-click the **Expenses.ExpenseRepository.WcfClient** project and select **Add | Existing Item…**. Navigate to the **AuthorizedExpenseServiceClient.cs** file provided with this lab and add it to the project.



1. Double-click the **AuthorizedExpenseServiceClient.cs** file to open it. This class inherits from our base **ExpenseServiceClient** and adds some behavior to the **CreateChannel** process. You can see the following lines where it establishes an operation context and adds our authorization header.

this.\_scope = new OperationContextScope(this.InnerChannel);

HttpRequestMessageProperty hrmp = new HttpRequestMessageProperty();

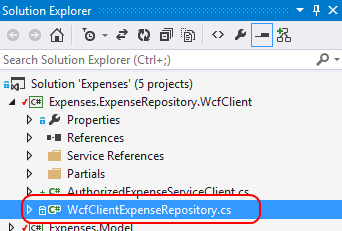
hrmp.Headers[HttpRequestHeader.Authorization] =

this.\_serviceAuthorizer.GetAuthorizationHeader();

OperationContext.Current.OutgoingMessageProperties[HttpRequestMessageProperty.Name] =

hrmp;

1. In the **Solution Explorer**, expand the **Expenses.ExpenseRepository.WcfClient** project and double-click **WcfClientExpenseRepository.cs** to open it.



1. Inside the class definition, add a private member to store an IServiceAuthorizer.

private IServiceAuthorizer \_serviceAuthorizer;

1. Next, update the public constructor to accept an IServiceAuthorizer parameter, which we’ll store in the private member we just created. You can replace the whole constructor with the code below.

public WcfClientExpenseRepository(string serviceUrl,

IServiceAuthorizer serviceAuthorizer)

{

this.\_serviceUrl = serviceUrl;

this.\_serviceAuthorizer = serviceAuthorizer;

}

1. Finally, update the **CreateExpenseServiceClient** method to use the new AuthorizedExpenseServiceClient. You can replace the whole method with the code below.

private WcfExpenseService.ExpenseServiceClient CreateExpenseServiceClient()

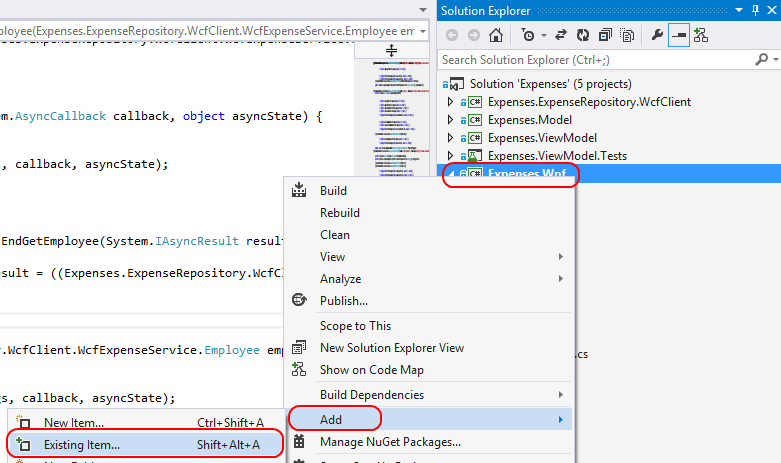
{

return new AuthorizedExpenseServiceClient(this.\_serviceUrl,

this.\_serviceAuthorizer);

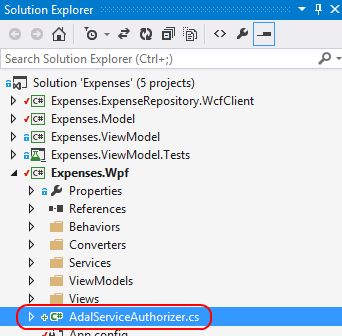
}

1. Now that we’ve updated the PCL components of our application, we can focus on the WPF client project. The first thing we’ll do is to integrate the Windows Azure Active Directory Authentication Library. Select the **Expenses.Wpf** project in the **Solution Explorer**.
2. Select **Tools | Library Package Manager | Package Manager Console**.
3. In the **Package Manager Console** dialog, type “Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory –version 1.0.2” and press **Enter**. This will install the Active Directory Authentication Library NuGet package.
4. There are three files we’re going to add to our project that take care of some of the utility functionality needed to acquire authentication tokens. Right-click the **Expenses.Wpf** project and select **Add | Existing Item…**. Navigate to the **AdalServiceAuthorizer.cs**, **CredManCache.cs**, and **CacheHelper.cs** files provided with this lab and add them to the project.

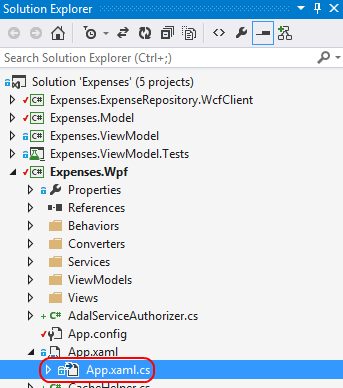


1. **CredManCache.cs** and **CacheHelper.cs** are helper files that support managing a local cache of credentials. We don’t need to make any changes there. **AdalServiceAuthorizer.cs**, on the other hand, is our implementation for **IServiceAuthorizer**. We’ll need to update some settings from our tenant client application before we can use it.

Double-click **AdalServiceAuthorizer.cs** in the **Solution Explorer** to open it.



1. Ordinarily we would want to have configuration settings for a class like this injected by an outside source, or at least loaded from a configuration file. For the sake of simplicity in this lab, we’re just going to use three private members.
   * Update **\_redirectUri** to use the **Redirect URI** you used when creating the tenant client application. In this lab it should be the URL to the root of your Azure Web Site based on how we’ve configured the initial settings.
   * Update **\_clientId** to use the **Client ID** you used when creating the tenant client application. It should be a string representation of a GUID.
   * Update **\_resourceBaseAddress** to use the **App ID URI** of the tenant service application. This is the same value we configured as the **audience** to be returned by our WCF service, which will be parsed by our WPF application during discovery. If these fields don’t match, we know something isn’t configured correctly, or that there is possibly a security threat.
2. Scroll down to the **AdalServiceAuthorizer** constructor. This is where we perform the initialization process, which includes an initial request to our secured service. Note that this initial request is expected to receive a “401 Unauthorized” response. However, it will contain the header information our application needs, such as the authorization URI.
3. Scroll down to the **GetAuthorizationHeader** method. This gets called by our SOAP client before it makes a request to the server so that we can acquire the appropriate authorization header. Close the file when ready to continue.
4. Since the **AdalServiceAuthorizer** class we added is our **IServiceAuthorizer**, we’ll use it to initialize the **IExpenseRepository** we use throughout the project. In the **Expenses.Wpf** project, double-click **App.xaml.cs** to open it.



1. Find the line where the **WcfClientExpenseRepository** is created and set into the **ServiceLocator**. Update it to look like the code below by adding the second parameter to the **WcfClientExpenseRepository** constructor.

ServiceLocator.Current.SetService<IExpenseRepository>(

new WcfClientExpenseRepository(url, new AdalServiceAuthorizer()));

1. Press **F5** to launch our updated WPF application. It will immediately present you with the Windows Azure Active Directory login. Note that you’ll need to use the credentials for the user you created earlier in our custom tenant. It will also ask you to change your password if this is the first login.
2. After a successful login, the application should work properly. Close when satisfied.

Note that if you run the application again, it may use your cached authorization and not ask you to log in. If this is undesirable behavior, such as due to computer sharing, it is possible to clear this behavior by calling **InternetSetOption** from **wininet.dll** to signal the end of the Internet session, thereby clearing the cache.

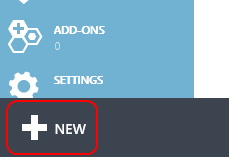
# Exercise 4: Integrating a cloud service with on-premises resources using Service Bus Relay

In this exercise, we’ll extend our WCF service to use an on-premises resource securely using Service Bus Relay.

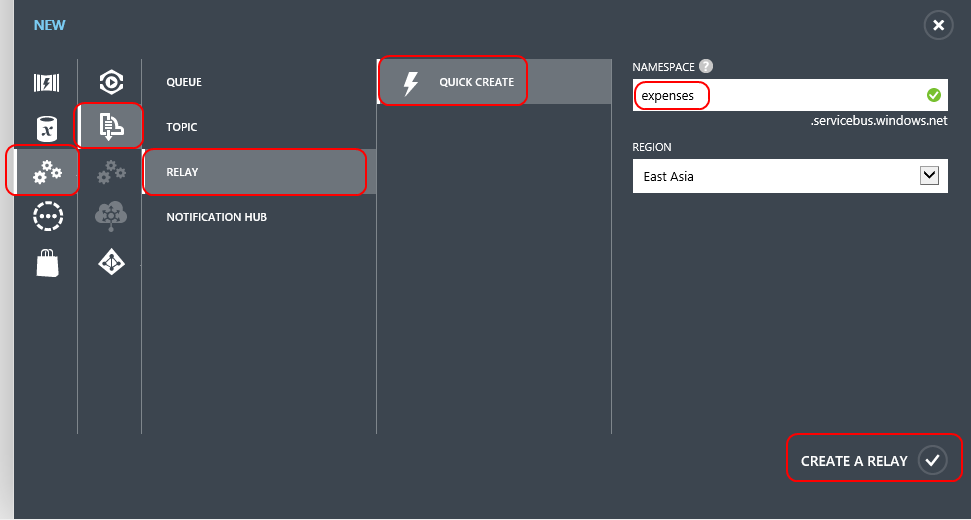
## Task 1: Creating a service bus relay

In this task, we’ll create a service bus relay.

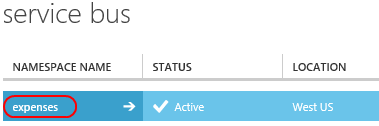
1. Log into your Azure administrative account at <https://manage.windowsazure.com>.
2. In the bottom right corner, click the **New** button.



1. Select **App Services | Service Bus | Relay | Quick Create** and enter a unique name for the **Namespace**. You’ll want to remember the **Namespace** for some work we’ll do later on. For the purposes of our demo, it doesn’t really matter when **Region** you select, but in the real world you should pick one in the same region as your application will be deployed. Click **Create a Relay** when ready.



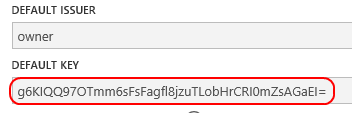
1. It may take a few moments for your namespace to become **Active**. Once it is active, click on it to see the details.



1. Near the bottom of the window, click the **Connection Information** button to view the connection details. It may take a few moments to appear. If it takes longer than ten seconds, you may need to dismiss the dialog and click the button again.



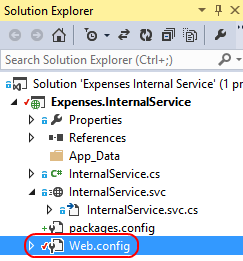
1. For the purposes of our lab, we only really need to take note of the **Default Key**. We can assume the **Default Issuer** is “owner”. We’ll need to copy the key for later use.



## Task 2: Add a service bus relay endpoint to the internal service

In this task, we’ll extend our internal service to listen on a service bus relay endpoint. Thanks to the power and flexibility of WCF, we’ll be able to make this update by referencing the service bus relay library and changing our Web.config. We won’t need to edit the code itself.

1. Open the **Expenses Internal Service\Expenses Internal Service.sln** solution.
2. Select the **Expenses.InternalService** project node in the **Solution Explorer**.
3. Select **Tools | Library Package Manager | Package Manager Console**.
4. In the **Package Manager Console** dialog, type “Install-Package WindowsAzure.ServiceBus –version 2.2.1.0” and press **Enter**. This will install the Windows Azure Service Bus NuGet package.
5. Double-click **Web.config** to open it.



1. If you scroll down to the **<extensions>** node, you’ll notice a lot of extensions registered by the service bus library. These allow us to reference the behaviors, transports, and bindings made available for our service. Note that if the extensions section doesn’t add references for items named “transportClientEndpointBehavior” or “netTcpRelayBinding”, then something didn’t work perfectly during the import. You should remove the NuGet package and then re-add it.
2. The first thing we’ll need to do is create an endpoint behavior that describes our service bus relay endpoint. Paste the following below the closing tag for the **</serviceBehaviors>** node. Update the **“YOUR DEFAULT KEY”** reference to use the key from the namespace we created earlier.

<endpointBehaviors>

<behavior name="sbTokenProvider">

<transportClientEndpointBehavior>

<tokenProvider>

<sharedSecret issuerName="owner" issuerSecret="YOUR DEFAULT KEY" />

</tokenProvider>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

1. Next, we’ll need to create the endpoint itself. Paste the following after the existing endpoint within the **Expenses.InternalService.InternalService** **<service>** node. Update the **“YOUR NAMESPACE”** reference to use the namespace you created earlier.

<endpoint contract="Expenses.InternalService.IInternalService"

binding="netTcpRelayBinding"

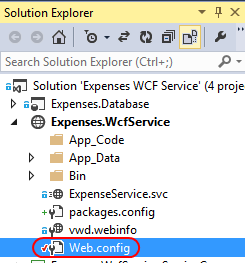
address="sb://YOUR NAMESPACE.servicebus.windows.net/internalservice"

behaviorConfiguration="sbTokenProvider"/>

## Task 3: Update the service client to use the service bus relay endpoint for the internal service

In this task, we’ll update our service client—the Expenses WCF Service—to use the service bus relay endpoint for the internal service. Note that this could just as easily be a WPF app or virtually any other kind of application.

1. Open the **Expenses WCF Service\Expenses WCF Service.sln** solution.
2. Select the **Expenses.WcfService** project node in the **Solution Explorer**.
3. Select **Tools | Library Package Manager | Package Manager Console**.
4. In the **Package Manager Console** dialog, type “Install-Package WindowsAzure.ServiceBus –version 2.2.1.0” and press **Enter**. This will install the Windows Azure Service Bus NuGet package.
5. Double-click **Web.config** to open it.



1. If you scroll down to the **<extensions>** node, you’ll notice a lot of extensions registered by the service bus library. These allow us to reference the behaviors, transports, and bindings made available for our service. Note that if the extensions section doesn’t add references for items named “transportClientEndpointBehavior” or “netTcpRelayBinding”, then something didn’t work perfectly during the import. You should remove the NuGet package and then re-add it.
2. The first thing we’ll need to do is create an endpoint behavior that describes our service bus relay endpoint. Paste the following below the closing tag for the **</serviceBehaviors>** node. Update the **“YOUR DEFAULT KEY”** reference to use the key from the namespace we created earlier.

<endpointBehaviors>

<behavior name="sbTokenProvider">

<transportClientEndpointBehavior>

<tokenProvider>

<sharedSecret issuerName="owner" issuerSecret="YOUR DEFAULT KEY" />

</tokenProvider>

</transportClientEndpointBehavior>

</behavior>

</endpointBehaviors>

1. Next, we’ll need to add the client endpoint for our requests to the internal service. Paste the following after the **</behaviors>** closing tag. Update the **“YOUR NAMESPACE”** reference to use the namespace you created earlier.

<client>

<endpoint name="internalservice"

contract="InternalService.IInternalService"

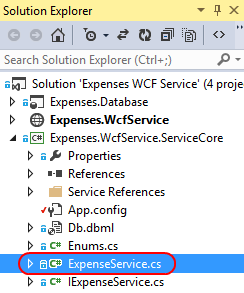
binding="netTcpRelayBinding"

address="sb://YOUR NAMESPACE.servicebus.windows.net/internalservice"

behaviorConfiguration="sbTokenProvider" />

</client>

1. Finally, we’ll need to update our service to use the internal service. Open the **ExpenseService.cs** file in the **Expenses.WcfService.ServiceCore** project.



1. In the **SaveExpenseReport** method, insert the following before the “return dbExpenseReport.ExpenseReportId” line (after the “db.SubmitChanges” line). This will notify the internal service when an expense report is saved with a status of “approved” for further processing.

if (dbExpenseReport.Status == (int)ExpenseReportStatus.Approved)

{

InternalServiceWrapper.ProcessReport(dbExpenseReport.ExpenseReportId);

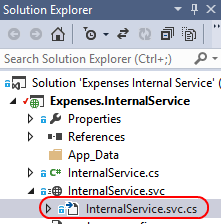
}

1. Publish the WCF service by right-clicking the project node and seleting **Publish Web Site**. Accept the defaults.

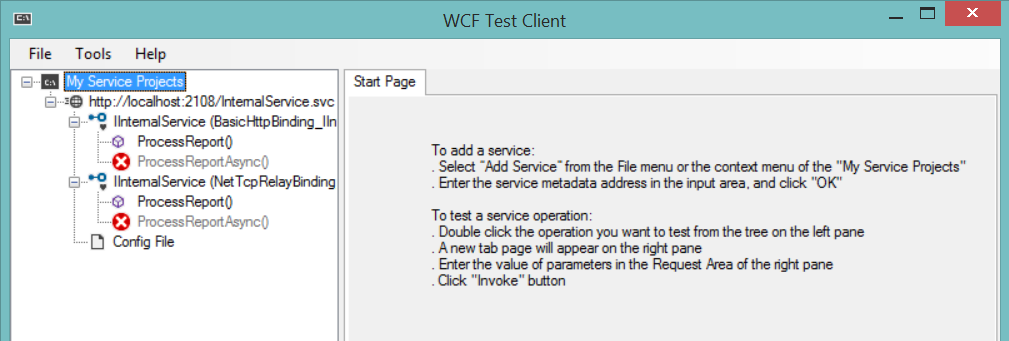
## Task 4: Verify the service bus relay integration

In this task, we’ll verify our service bus relay integration.

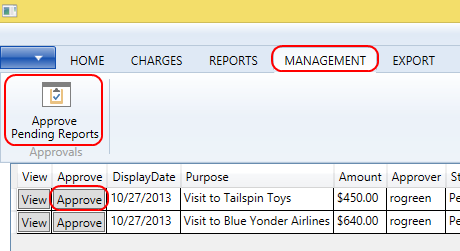
1. In the **Expenses Internal Service** solution, open the **InternalService.svc.cs** file.



1. Set a breakpoint inside the **ProcessReport** method.
2. Press **F5** to begin a debug session. The service will be hosted in the WCF Test Client and will now listen on our service bus endpoint.



1. Open **\Expenses\Expenses.sln** in Visual Studio 2013. Use a new instance of Visual Studio so that both the service and client solutions are open.
2. Press **F5** to launch the **Expenses WPF** app. Select the **Management** tab and click **Approve Pending Reports**. Click **Approve** on one of the pending reports to kick off the approval process.



1. If everything executed successfully, your internal service should hit the breakpoint that was invoked by the service relay request (via the Azure Web site hosting our WCF service). You can mouse over the **expenseReportId** parameter to see which one it was.